Unit - 4
Recursion

#Recursion:

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

For the problems solve recursively following two conditions must be satisfied:

(1) Each time a function calls itself and it must be closer to a ~~function~~ solution.

(2) The problem statement must include a stopping condition.

Example: Finding factorial of a given number.
Let us consider the factorial of a number and its algorithm described recursively:
We know that,

$n! = n*(n-1)!$

$(n-1)! = n-1*(n-2)!$ And so on up to 1.

Algorithm:

```
Factorial (n)
{
    if n==1
        return 1
    else
        return n* Factorial(n-1)
}
```

Let's trace the evaluation of factorial(5)

Factorial(5) =
5 * factorial(4) =
5 * (4 * factorial(3)) =
5 * (4 * (3 * factorial(2))) =
5 * (4 * (3 * (2 * factorial(1)))) =
5 * (4 * (3 * (2 * (1 * factorial(0))))) =
5 * (4 * (3 * (2 * (1 * 1)))) =
5 * (4 * (3 * (2 * 1))) =
5 * (4 * (3 * 2)) =
5 * (4 * 6) =
5 * 24 =
120

Divide and Conquer Algorithms:

A divide and conquer algorithm is an effieient recursive algorithm that consists of two parts:

- Divide, in which smaller problems are solved recursively (except, of course, base cases)
- Conquer, in which the solution to the original problem is then formed from the solutions to the sub problems.

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms.

Key differences between recursion and iteration:

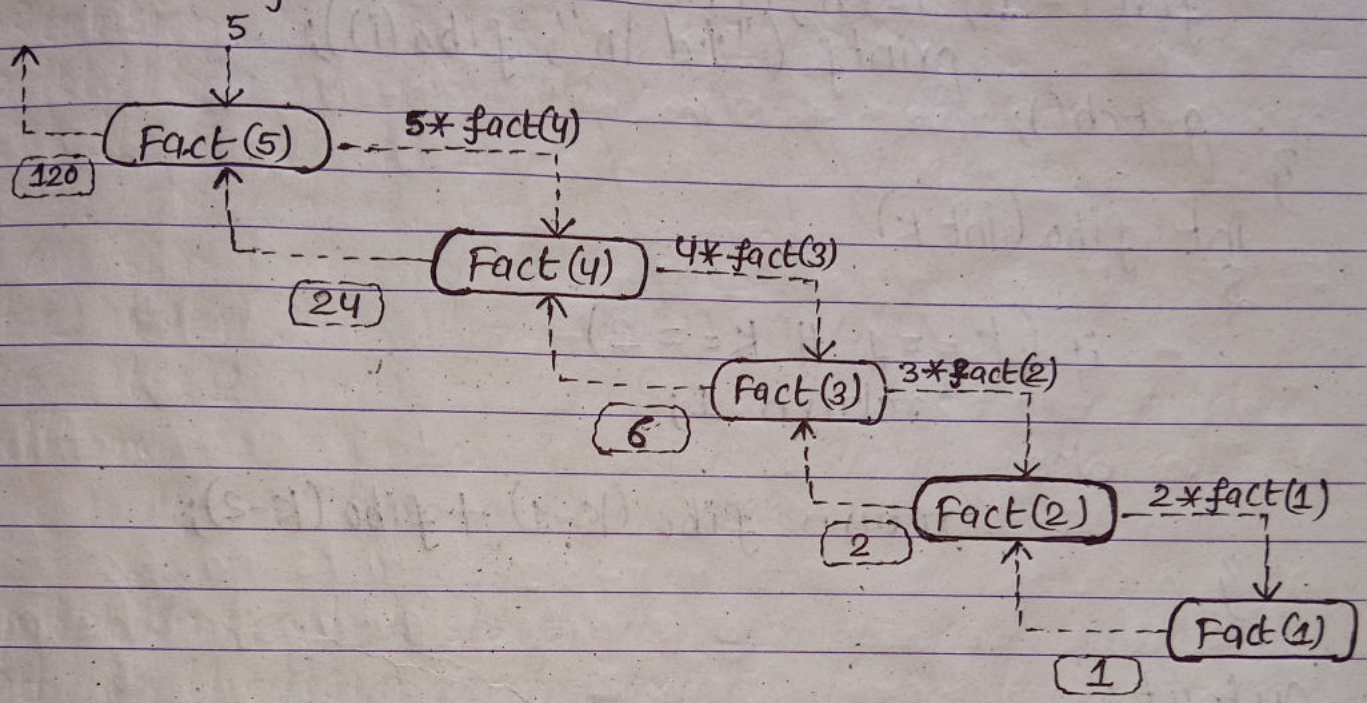
Recursion	Iteration
(1) Recursion is when a method in a program repeatedly calls itself.	(1) Iteration is when a set of instructions in a program are repeatedly executed.
(2) A recursive method contains set of instructions, statement calling itself, and a termination condition.	(2) Iteration statements contain initialization, increment, condition, set of instruction within a loop and a control variable.
(3) If the method does not lead to the termination condition it enters to infinite recursion.	(3) If the control variable never leads to the termination value the iteration statement iterates infinitely.
(4) Infinite recursion can lead to system crash.	(4) Infinite iteration consumes CPU cycles.
(5) Variables created during recursion are stored on stack memory.	(5) Iteration doesn't require a stack.
(6) Due to function calling overhead execution of recursion is slower.	(6) Execution of iteration is faster.
(7) Recursion reduces the size of code.	(7) Iteration make a code longer.

Recursion Examples:

Example 1: Calculation of the factorial of an integer number using recursive function.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n;
    long int facto;
    long int factorial (int n);
    printf ("Enter value of n:");
    scanf ("%d", &n);
    facto = factorial (n);
    printf ("%d! = %ld", n, facto);
    getch();
}
long int factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial (n-1);
}
```

Output:
Enter value of n:
6
Factorial of 6 = 720



Example 2: Program to generate Fibonacci series up to n terms using recursive function.
(Hint Fibonacci sequence = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n, i;
    int fibo(int);
```

```
    printf("Enter value of n:");
    scanf("%d",&n);
    printf("Fibonacci numbers up to %d terms:\n",n);
    for(i=1; i<=n; i++)
            printf("%d \n", fibo(i));
    getch();
}
int fibo(int k)
{
    if(k==1 || k==2)
        return 1;
    else
        return fibo(k-1)+fibo(k-2);
}
```

Output:

```
Enter value of n:
7
Fibonacci numbers up to 7th terms:
1 1 2 3 5 8 13
```

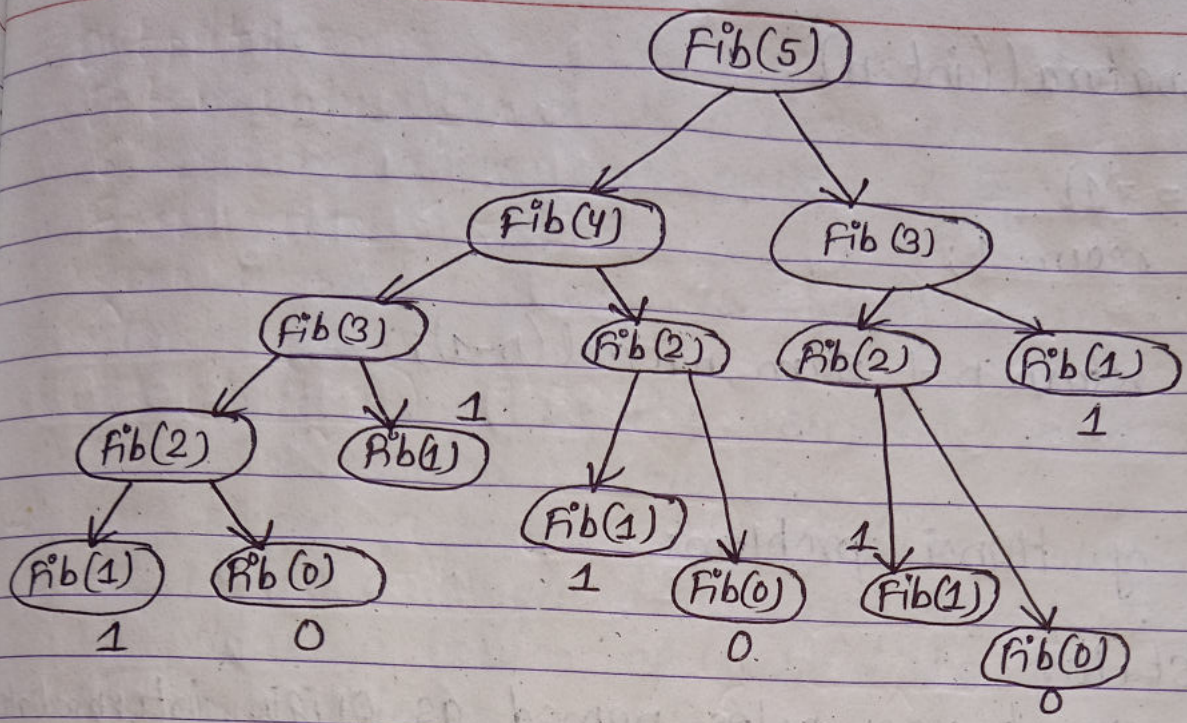


Fig: 5th term of Fibonacci series = 1+0+1+1+0+1+0+1 = 5

Example 3: Program to find sum of first n natural numbers using recursion.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n;
    int sum_natural(int);
    printf("Enter value of n:");
    scanf("%d", &n);
    printf("Sum of first %d natural numbers = %d",
      n, sum_natural(n));
    getch();
}
```

```
int sum_natural(int n)
{
    if(n==1)
        return 1;
    else
        return n + sum_natural(n-1);
}
```

Tower of Hanoi problem:

Initial state:

↳ There are three poles named as origin, intermediate and destination.

↳ n number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.

↳ The disks are numbered as 1, 2, 3, 4---n.

Objective:

↳ Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions:

↳ Move only one disk at a time.

↳ Each disk must always be placed around one of the pole.

↳ Never place larger disk on top of smaller disk.

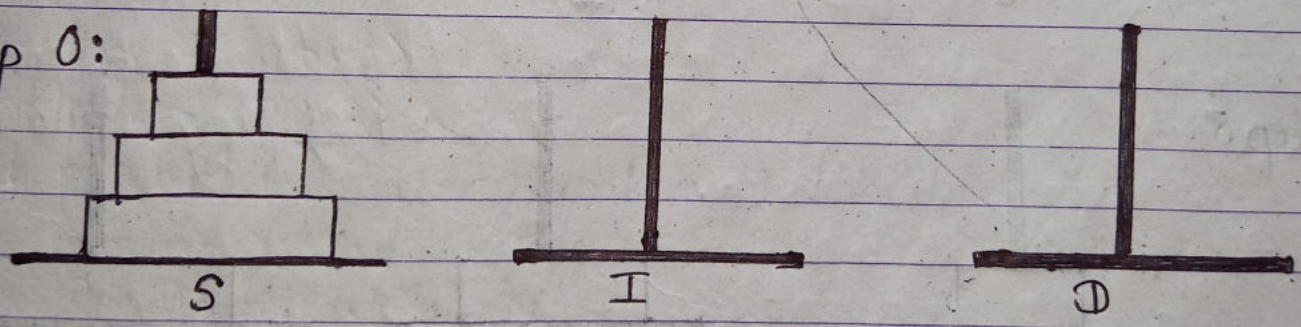
Algorithm:

To move a tower of n disks from source to destination (where n is positive integer)

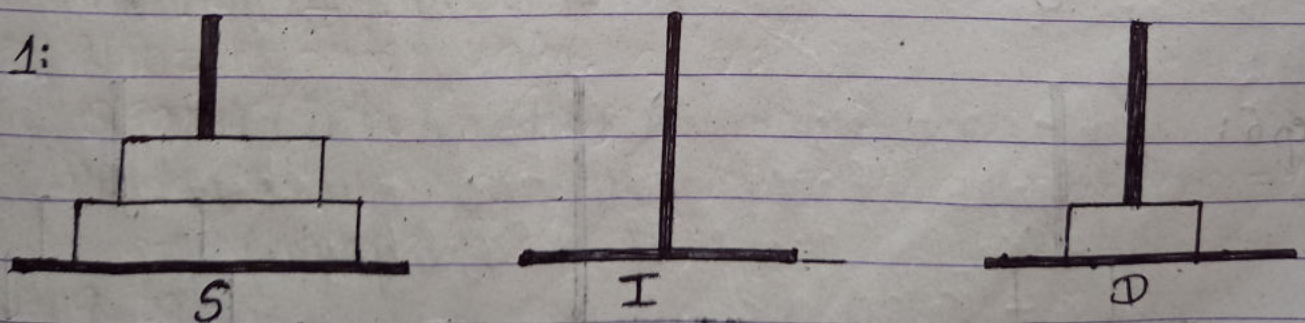
1. If n==1:
   1.1 Move a single disk from source to destination
2. If n> 1:
   2.1. Let temp be the remaining pole other than source and destination
   2.2. Move a tower of (n-1) disks from source to temp
   2.3. Move a single disk from source to destination.
   2.4. Move a tower of (n-1) disks form temp to destination
3. Stop

Tracing (Tracing for n=3)

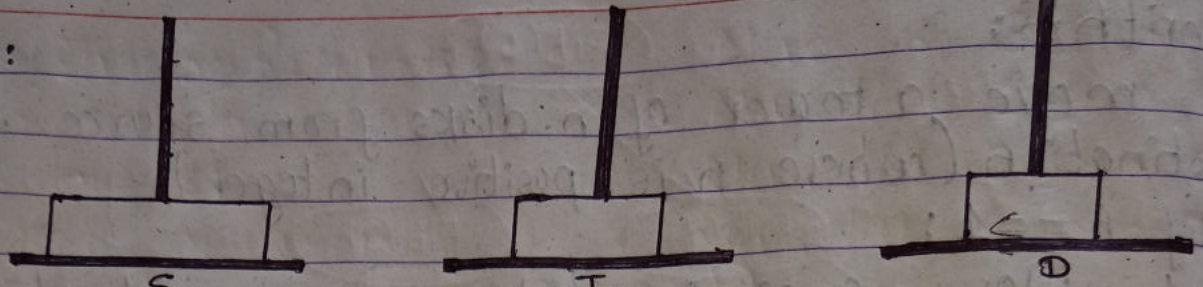
Step 0:



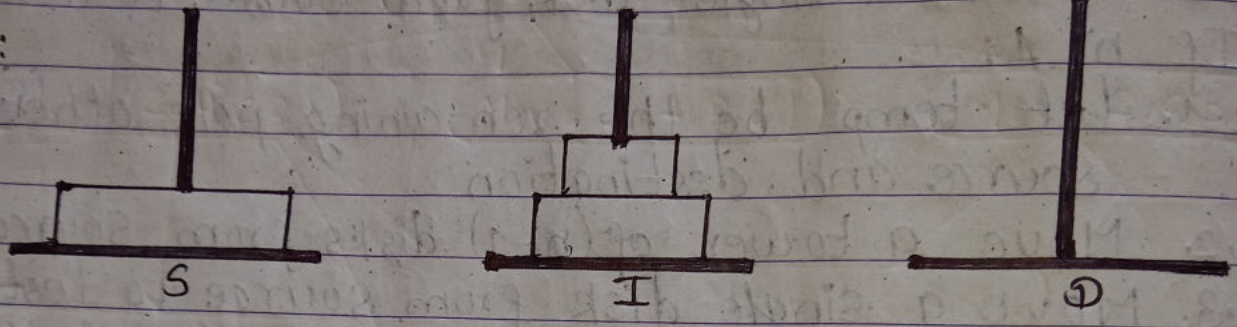
Step 1:



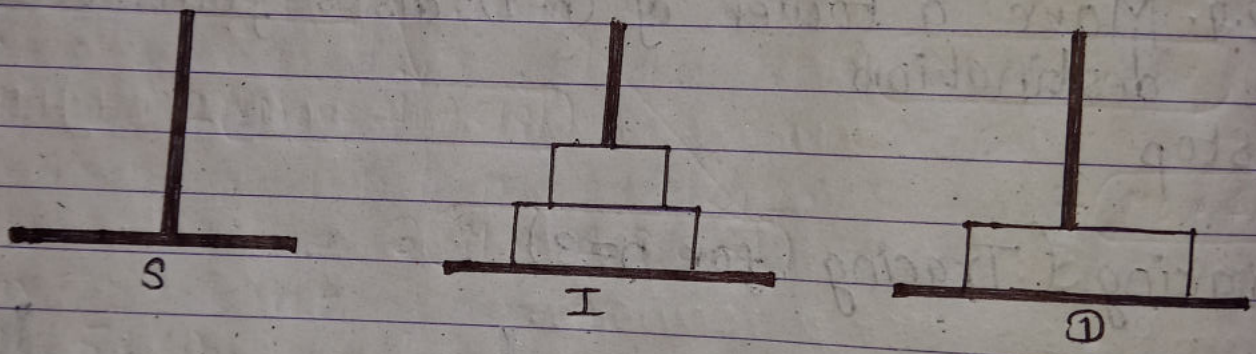
Step 2:



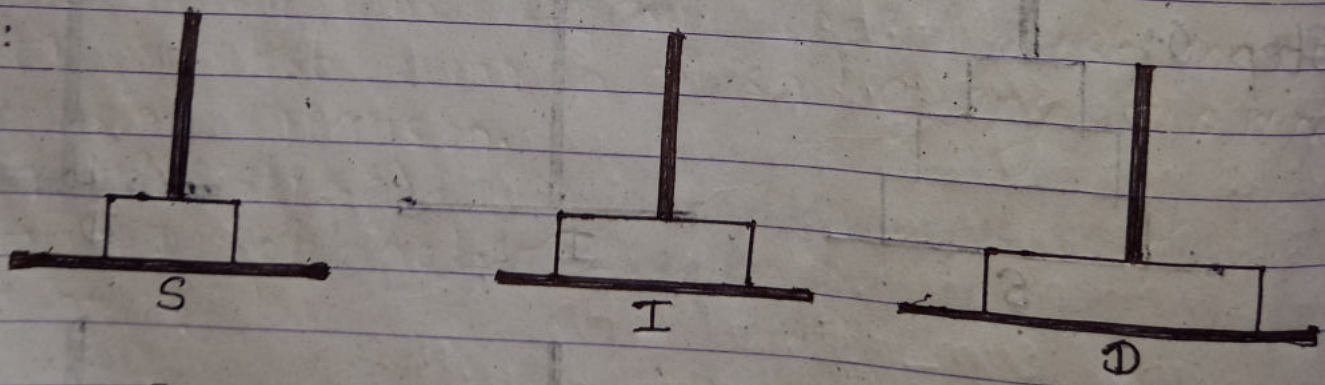
Step 3:



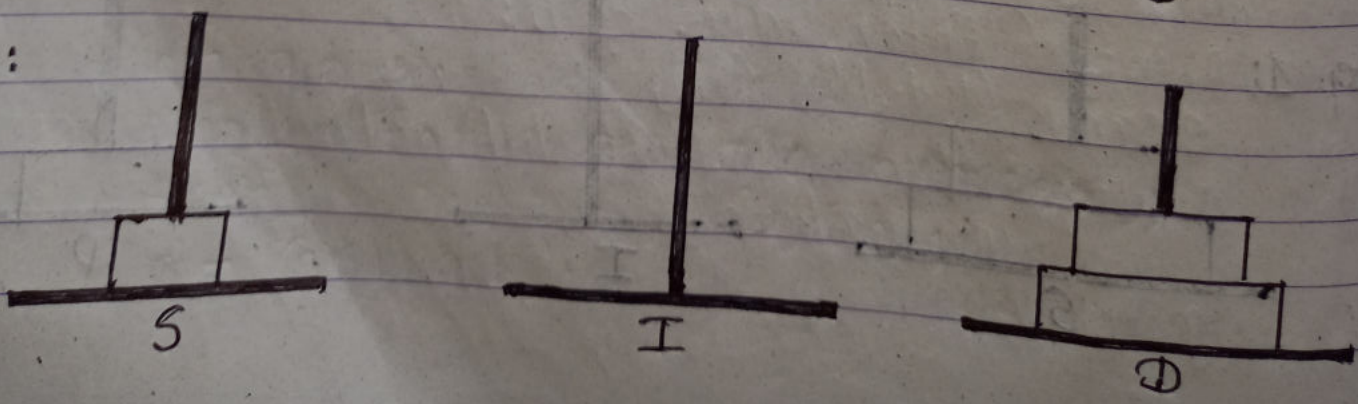
Step 4:



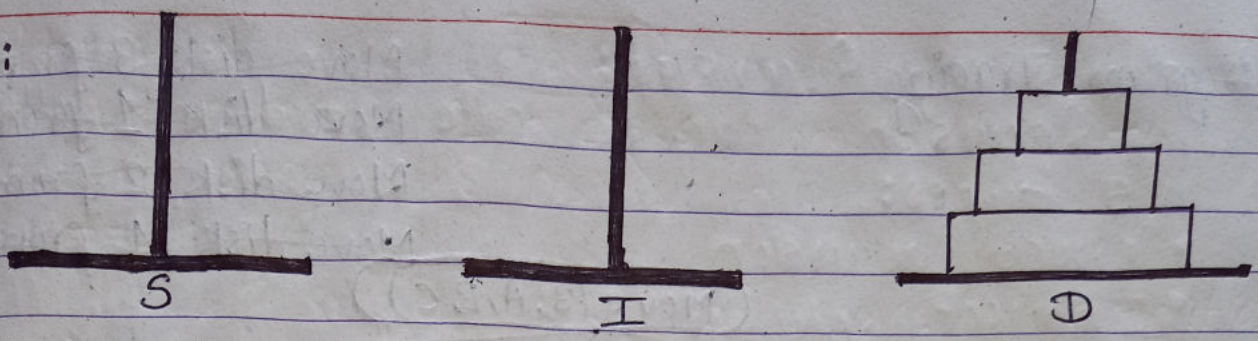
Step 5:



Step 6:



Step 7:



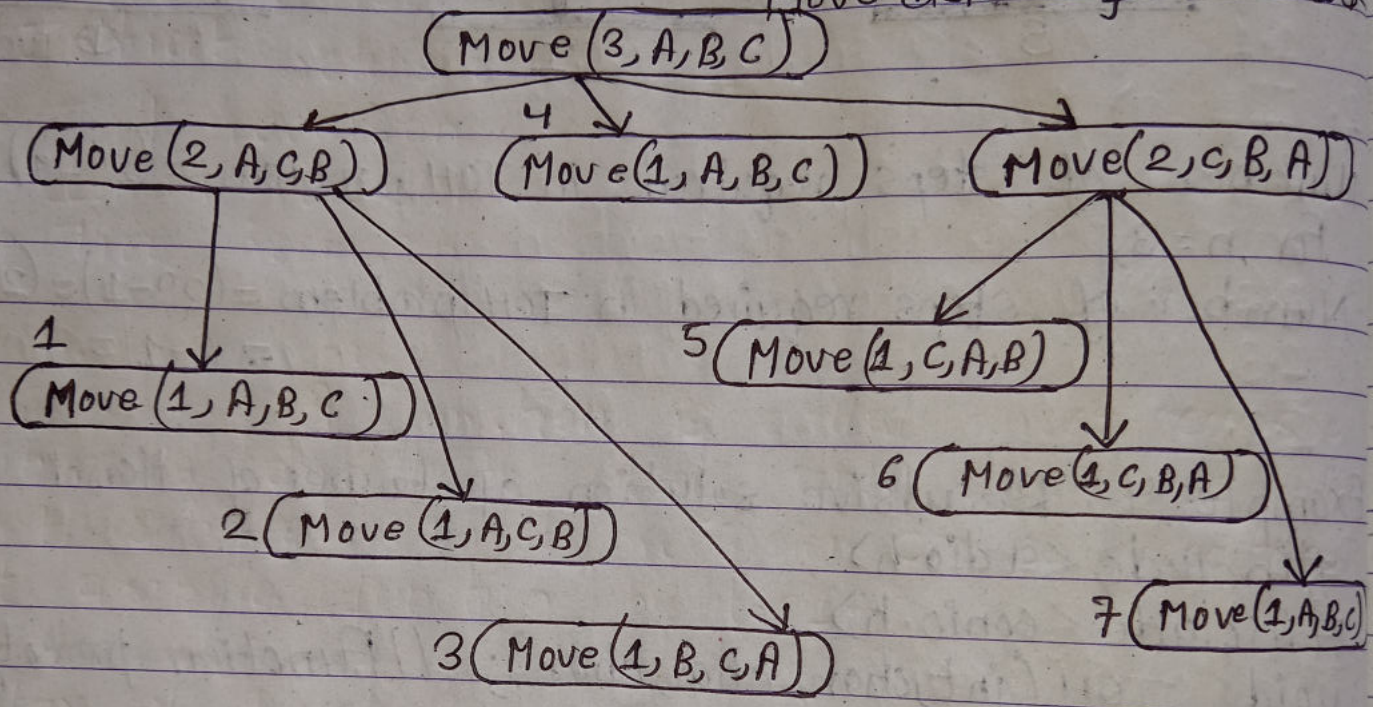
Number of steps required in TOH problem $= (2^n - 1)$
for $n = 3$,
Number of steps required in TOH problem $= (2^n - 1) = (2^3 - 1)$
$= 8 - 1 = 7$

Example 4: Recursive solution of tower of Hanoi

```
#include <stdio.h>
#include <conio.h>
void TOH (int,char, char, char); //Function prototype
void main()
{
    int n;
    printf ("Enter number of disks");
    scanf ("%d", &n);
    TOH (n, 'O', 'D', 'I');
    getch();
}
void TOH (int n, char A, char B, char C)
{
    if (n>0)
    {
    TOH (n-1, A, C, B);
    printf ("Move disk %d from %c to %c \n", n, A, B);
    TOH (n-1, C, B, A);
}}
```

Program Tracing

~~Move disk 3 from I to~~
~~Move disk 1 from O to~~
~~Move disk 2 from O to~~
~~Move disk 1 from I to~~



Example 5: Program to find the Highest common factor (greatest common divisor) of any two numbers by using recursion.

```
#include <stdio.h>
int GCD (int, int);
int main()
{
    int a, b, g;
    printf ("Enter any two numbers");
    scanf ("%d %d", &a, &b);
    g = GCD(a,b);
```

```
printf("GCD of given two numbers=%d", g);
return 0;
}
int GCD(int x, int y)
{
    if(y==0)
        return x;
    else
        return(GCD(y, x%y));
}
```

#Advantages of Recursion:

↳ The code may be much easier to write.

↳ To solve some problems which are naturally recursive such as tower of Hanoi.

#Disadvantages of Recursion:

↳ Recursive functions are generally slower than non-recursive functions.

↳ May require a lot of memory to hold intermediate results on the system stack.

↳ It is difficult to think recursively so one must be very careful when writing recursive functions.

Types of Recursion:

Recursion can be categorized into following 5 types:

- Direct recursion
- Indirect recursion
- Tail recursion
- Linear recursion
- Tree recursion

- **Direct recursion:**

A function is called direct recursive if it calls directly to itself until some base condition is not satisfied. Direct recursion can be used to call just a single function by itself.

Syntax of Direct Recursion:

```
Return type function_name(p1, p2, --, pn)
{
    ----------
    ----------
    Function_name(p1, p2, --, pn);
    ----------
    ----------
}
```

Example: Function for finding factorial of given number by using direct recursion method.

```
int fact (int num)
{
    if(num==0)
        return 1;
    else
        return num*fact(num-1);
}
```

- Indirect recursion:

A function is said to be indirect recursive if it calls to another function and again this new function calls to its calling function.

Example: Program for Indirect recursion in C

```
int fun1(int x)
{
    if (x<=0)
        return 1;
    else
        return fun2(x);
}
int fun2 (int y)
{
    return fun1(y-1);
}
```

What is the difference between direct and indirect recursion?

⇒ A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in below.

An example of direct recursion

```
void directRecFun()
{
    // some code...
    directRecFun();
    // some code...
}
```

An example of indirect recursion

```
void indirectRecFun1()
{
    // some code...
    indirectRecFun2();
    // some code...
}
void indirectRecFun2()
{
    // some code...
    indirectRecFun1();
    // some code...
}
```

8. Tail recursion:

A function that returns the value of its recursive call is said to be tail recursive. Simply a function is said to be tail recursive if there are no any calculations occur in the recursive stage and it only returns the value.

Example: Complete program in C to showing the use of tail recursion.

```
Fact (n, accumulator)
{
    if(n==0)
        return accumulator;
    else
        return Fact (n-1, n*accumulator);
}
factorial (n)
{
    return Fact(n,1);
}
void main()
{
    int num, f;
    printf ("Enter any number");
    scanf ("%d", &num);
    f=factorial (num);
```

```
    printf("Factorial of given number is %d", f);
    getch();
}
```

Output:

Enter any number
10
Factorial of given number = 3628800